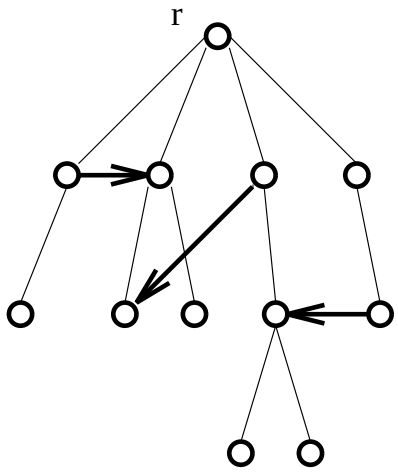
BFS Trees

If BFS is performed on a connected, undirected graph, a tree is defined by the edges involved with the discovery of new nodes:



This tree defines a shortest path from the root to every other node in the tree.

The proof is by induction on the length of the shortest path from the root:

- *Length = 1* First step of BFS explores all neighbors of the root. In an unweighted graph one edge must be the shortest path to any node.

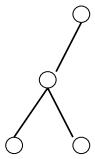
- *Length = s* Assume the BFS tree has the shortest paths up to length $s-1$. Any node at a distance of s will first be discovered by expanding a distance $s-1$ node.

Edge Classification for DFS

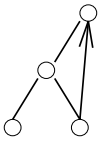
What about the other edges in the graph? Where can they go on a search?

Every edge is either:

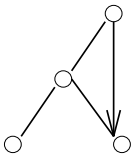
1. A Tree Edge



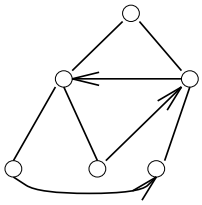
2. A Back Edge to an ancestor



3. A Forward Edge to a decendant



4. A Cross Edge to a different node



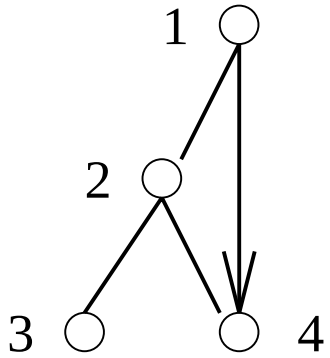
On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

DFS Trees

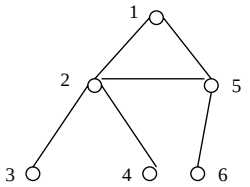
The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.

In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

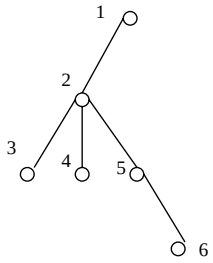
Why? Suppose we have a forward edge. We would have encountered $(4, 1)$ when expanding 4, so this is a back edge.



Suppose we have a cross-edge

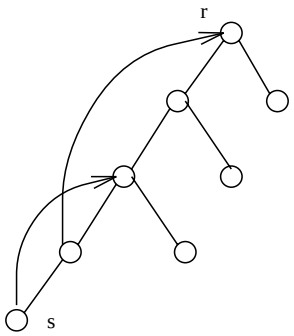


When expanding 2, we would discover 5, so the tree would look like:



Paths in search trees

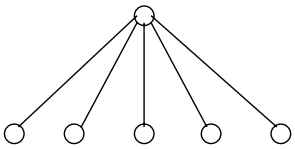
Where is the shortest path in a DFS?



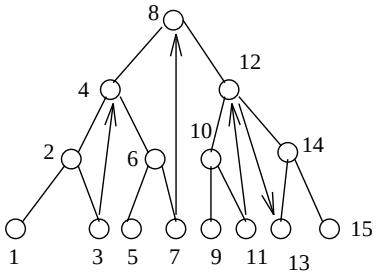
It could use multiple back and tree edges, where BFS only used tree edges.

It could use multiple back and tree edges, where BFS only uses tree edges.

DFS gives a better approximation of the longest path than BFS.



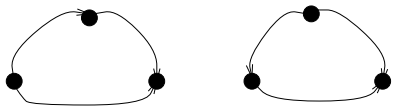
The BFS tree can have height 1, independant of the length of the longest path.



The DFS must always have height >= log P, where P is the length of the longest path.

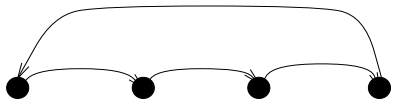
Topological Sorting

A directed, acyclic graph is a directed graph with no directed cycles.



A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.

Only a DAG can have a topological sort.



Any DAG has (at least one) topological sort.

Applications of Topological Sorting

Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given constraints, such as a set of left-right constraints on the positions of objects.

Example: Dressing schedule from CLR.

Example: Identifying errors in DNA fragment assembly.

Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.

```
A B R A C
A C A D A
A D A B R
D A B R A
R A C A D
```

```
A B R A C A D A B R A
A B R A C
    R A C A D
        A C A D A
            A D A B R
                D A B R A
```

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistant ordering. If there are cycles, there must be errors.

A DFS can test if a graph is a DAG (it is iff there are no back edges - forward edges are allowed for DFS on directed graph).

Algorithm

Theorem: Arranging vertices in decreasing order of DFS finishing time gives a topological sort of a DAG.

Proof: Consider any directed edge u, v, when we encounter it during the exploration of vertex u:

- If v is white - we then start a DFS of v before we continue with u.
- If v is grey - then u, v is a back edge, which cannot happen in a DAG.
- If v is black - we have already finished with v, so $f[v] < f[u]$.

Thus we can do topological sorting in $O(n + m)$ time.